

Toward a Succinct Index for Order-Preserving Pattern Matching

Travis Gagie¹ and Giovanni Manzini^{2,3}

¹ School of Computer Science and Telecommunications, Diego Portales University, Santiago, Chile

² Computer Science Institute, University of Eastern Piedmont, Alessandria, Italy

³ IIT-CNR, Pisa, Italy

Abstract. Although introduced only a few years ago, the problem of order-preserving pattern matching has already attracted significant attention. In this paper we introduce the first index for this problem that uses sublinear space in addition to the data, which is useful when we want to search repeatedly in datasets to which we have read-only access and which are too large to copy conveniently. Specifically, we show how, given a string $S[1..n]$ and a constant $c \geq 1$, we can build an $\mathcal{O}(n \log \log n)$ -bit index such that later, given a pattern $P[1..m]$ with $m \leq \lg^c n$ and fast random access to S , we can return the starting position of some order-preserving match for P in S in $\mathcal{O}(m \log^4 n)$ time if one exists.

1 Introduction

Mining for correlations in large datasets is complicated by amplification or damping: e.g., the euro fluctuating against the dollar may cause the pound to fluctuate similarly a few days later, but to a greater or lesser extent. If we search only for sequences of values that rise or fall in exactly the same way as those in a pattern, therefore, either relatively or absolutely, we are likely to miss many potentially interesting leads. Searching for sequences in which only the relative order of the values is constrained to be the same as for the pattern is more robust. Kubica et al. [20] and Kim et al. [19] formally introduced this problem under the name *order-preserving pattern matching* and gave efficient online algorithms for it. For example, in 6, 3, 9, 2, 7, 5, 4, 8, 1, the order-preserving matches of 2, 1, 3 are 6, 3, 9 and 5, 4, 8. Several sets of authors have continued their investigation, and we briefly survey their results in Section 2.

Crochemore et al. [12] showed how, given a string $S[1..n]$, in $\mathcal{O}(n \log(n) / \log \log n)$ time we can build an $\mathcal{O}(n \log n)$ -bit index such that later, given a pattern $P[1..m]$, we can return the starting positions of all the occ order-preserving matches of P in S in optimal $\mathcal{O}(m + \text{occ})$ time. Their index is a kind of suffix tree, and other researchers [21] are trying to reduce the space bound to $n \lg \sigma + o(n \log \sigma)$ bits, where σ is the size of the alphabet of S , by using a kind of Burrow-Wheeler Transform instead. Even if they succeed, however, when $\sigma = \Omega(n)$ the resulting index will still take linear space — i.e., $\Omega(n)$ words or $\Omega(n \log n)$ bits.

An index taking $n \lg \sigma + o(n \log \sigma)$ bits is often called *succinct* because its space usage differs from the information-theoretic minimum by only a lower-order term. However, some researchers (see, e.g., [3]) draw a distinction between *succinct encodings*, which can change the underlying data (by applying the Burrows-Wheeler Transform, for instance) and mix it with their auxiliary data structures, and *succinct indexes*, which use read-only access to the underlying data and keep their auxiliary data structures separate. The latter have several advantages: e.g., several succinct indexes can use the the same underlying data, and the dataset may be publicly available to read but we may have to copy it if we want to manipulate it, which may be inconvenient if it is large.

In this paper we show how, given $S[1..n]$ and a constant $c \geq 1$, we can build an $\mathcal{O}(n \log \log n)$ -bit index such that later, given $P[1..m]$ with $m \leq \lg^c n$ and fast random access to S , we can return

the starting position of some order-preserving match for P in S in $\mathcal{O}(m \log^4 n)$ time if one exists. By “fast random access” we mean that we can read any ℓ consecutive characters of S in $\mathcal{O}(\ell \log n)$ time for any ℓ . When σ is superpolylogarithmic in n , i.e., $\sigma = \log^{\omega(1)} n$, this is a succinct index, and in any case we always use sublinear space (measured in words) in addition to the data.

2 Previous Work

Although recently introduced, order-preserving pattern matching has received considerable attention and has been studied in different settings. For the online problem, where the pattern is given in advance, the first contributions were inspired by the classical Knuth-Morris-Pratt and Boyer-Moore algorithms [4,11,19,20]. The proposed algorithms have guaranteed linear time worst-case complexity or sublinear time average complexity. However, for the online problem the best results in practice are obtained by algorithms based on the concept of filtration, in which some sort of “order-preserving” fingerprint is applied to the text and the pattern [5,6,7,9,10,14]. This approach was successfully applied also to the harder problem of matching with errors [7,17,18].

For the offline problem, when the text is given in advance, in addition to the already mentioned results by Crochemore et al. [12] other solutions have been proposed combining the idea of fingerprint and indexing. In [8] the authors show how speedup the search building an FM-index [15] on the binary string expressing whether in the input text each element is smaller or larger than the next one. By expanding this approach, [13] show how to build a compressed file format supporting order-preserving matching without the need of full decompression. Experiments show that this compressed file format takes roughly the same space as `gzip` and that in most cases the search is orders of magnitude faster than the sequential scan of the text. We point out that the approaches in [8,13], although interesting for the applications, do not have competitive worst case bounds on the search cost as the one in [12] and in the present paper.

3 Our Index

Given a sequence $S[1..n]$ we define the rank encoding $E(S)[1..n]$ as

$$E(S)[i] = \begin{cases} 0.5 & \text{if } S[i] \text{ is lexicographically smaller than any} \\ & \text{character in } \{S[1], \dots, S[i-1]\}, \\ j & \text{if } S[i] \text{ is equal to the lexicographically } j\text{th} \\ & \text{character in } \{S[1], \dots, S[i-1]\}, \\ j + 0.5 & \text{if } S[i] \text{ is larger than the lexicographically } j\text{th} \\ & \text{character in } \{S[1], \dots, S[i-1]\} \text{ but smaller than} \\ & \text{the lexicographically } (j+1)\text{st}, \\ |\{S[1], \dots, S[i-1]\}| + 0.5 & \text{if } S[i] \text{ is lexicographically larger than any} \\ & \text{character in } \{S[1], \dots, S[i-1]\}. \end{cases}$$

This is similar to the encodings used in previous papers on order-preserving matching. We can build $E(S)$ in $\mathcal{O}(n \log n)$ time and $E(S) = E(S')$ if and only if S and S' are an order-preserving match.

We define the rank-encoded suffix array $R[1..n]$ of S such that $R[i] = j$ if $E(S[i..n])$ is the lexicographically j th string in $\{E(S), E(S[2..n]), \dots, E(S[n])\}$. Note that $E(S[i..n])$ has length $n - i + 1$. Figure 1 shows an example. Similarly to how we can use a normal suffix array to support normal pattern matching, we can use R to find all order-preserving matches for a pattern $P[1..m]$ in $\mathcal{O}(m \log^2 n)$ time via binary search — i.e., at each step we choose an index i , extract $S[R[i]..R[i] + m - 1]$, compute its rank encoding and compare it to $E(P)$, all in $\mathcal{O}(m \log n)$ time — but storing R takes $\Omega(n \log n)$ bits. Therefore, we sample and store only every $\lfloor \lg n \rfloor$ th element of R and every element of R that is 1 or n a multiple of $\lfloor \lg n \rfloor$, which takes only $\mathcal{O}(n)$ bits. Notice we can still find in $\mathcal{O}(m \log^2 n)$ time via binary search in R an ordering-preserving match for any pattern $P[1..m]$ that has at least $\lfloor \lg n \rfloor$ order-preserving matches in S . If P has fewer than $\lfloor \lg n \rfloor$ order-preserving matches in S but we happen to have sampled a cell of R pointing to the starting position of one, then our binary search still finds it; otherwise, we find an interval of length at most $\lfloor \lg n \rfloor - 1$ which contains pointers to all the order-preserving matches for P in S .

Suppose that for each unsampled element $R[i] = j$ we store the following data:

- the smallest number $L[i]$ (if one exists) such that $S[j - 1..j + L[i] - 1]$ has at most $\lg^c n$ order-preserving matches in S ;
- the rank $B[i] = E[L[i] + 1](S[j - 1..j + L[i] - 1]^{\text{rev}}) \leq L[i] + 1/2$ of $S[j - 1]$ in $S[j..j + L[i] - 1]$, where the superscript rev indicates that the string is reversed;
- the distance $D[i] < \lg^c n + \lg n$ to the cell of R containing $j - 1$ from the last sampled element x such that $E(S[x..x + L[i]])$ is lexicographically smaller than $E(S[j - 1..j + L[i] - 1])$.

Figure 1 shows the values in L , B and D for our example.

4 Searching

Assume we are given $P[1..m]$ and i and told that $S[R[i]..R[i] + m - 1]$ is an order-preserving match for P , but we are not told the value $R[i] = j$. If $R[i]$ is sampled, of course, then we can return j immediately. If $L[i]$ does not exist or is greater than m then P has at least $\lg^c n \geq \lfloor \lg n \rfloor$ order-preserving matches in S , so we can find one in $\mathcal{O}(m \log^2 n)$ time with R . Otherwise, from $L[i]$, $B[i]$ and P , we can compute $E(S[j - 1..j + L[i] - 1])$ in $\mathcal{O}(m \log m)$ time: we take the length- $L[i]$ prefix of P ; if $B[i]$ is an integer, we prepend to $P[1..L[i]]$ a character equal to the lexicographically $B[i]$ th character in that prefix; if $B[i]$ is $r + 0.5$ for some integer r with $1 \leq r < L[i]$, we prepend a character lexicographically between the lexicographically r th and $(r + 1)$ st characters in the prefix; if $B[i] = 0.5$ or $B[i] = L[i] + 0.5$, we prepend a character lexicographically smaller or larger than any in the prefix, respectively. We can then find in $\mathcal{O}(m \log^2 n)$ time the position in R of x , the last sampled element such that $E(S[x..x + L[i]])$ is lexicographically smaller than $E(S[j - 1..j + L[i] - 1])$. Adding $D[i]$ to this position gives us the position i' of $j - 1$ in R . Repeating this procedure until we reach a sampled cell of R takes $\mathcal{O}(m \log^3 n)$ time, and we can then compute and return j . As the reader may have noticed, this is very similar to how we use backward stepping to locate occurrences of a pattern with an FM-index [15].

Even if we do not really know whether $S[R[i]..R[i] + m - 1]$ is an order-preserving match for P , we can still start at the cell $R[i]$ and repeatedly apply this procedure: if we do not find a sampled cell after $\lfloor \lg n \rfloor - 1$ repetitions, then $S[R[i]..R[i] + m - 1]$ is not an order-preserving match for P ; if we do, then we add the number of times we have repeated the procedure to the contents of the

i	$R[i]$	$L[i]$	$B[i]$	$D[i]$	$E(S[R[i]..n])$
1	30				0.5
2	29	2	1.5	4	0.5 0.5
3	22	2	0.5	2	0.5 0.5 0.5 0.5 1.5 5 5.5 6.5 1
4	13				0.5 0.5 0.5 1 0.5 1.5 4 4.5 1 4 3.5 3.5 2 3 6 7 7.5 2
5	2	2	0.5	1	0.5 0.5 0.5 1.5 2.5 3.5 5.5 2.5 2 5 4 8 4 1 1 0.5 1.5 6 7 1 6 5 4 2 3 6 7 8 2
6	23	3	3.5	3	0.5 0.5 0.5 1.5 4.5 5.5 6.5 1
7	8				0.5 0.5 0.5 2.5 2.5 5.5 3 0.5 1 0.5 1.5 6 7 1 6 5 4 2 3 6 7 7.5 2
8	14				0.5 0.5 1 0.5 1.5 4 4.5 1 4 3.5 3.5 2 3 6 7 7.5 2
9	20				0.5 0.5 1.5 1.5 1.5 1.5 2.5 6 7 7.5 2
10	3	3	3.5	1	0.5 0.5 1.5 2.5 3.5 5.5 2.5 2 5 4 7.5 4 1 1 0.5 1.5 6 7 1 6 5 4 2 3 6 7 8 2
11	16				0.5 0.5 1.5 3.5 4.5 1 4 3.5 3.5 2 3 6 7 7.5 2
12	24				0.5 0.5 1.5 3.5 4.5 5.5 1
13	11	2	0.5	3	0.5 0.5 2.5 1 0.5 1 0.5 1.5 4 5 1 4 3.5 3.5 2 3 6 7 7.5 2
14	9	3	3.5	3	0.5 0.5 2.5 2.5 4.5 3 0.5 1 0.5 1.5 6 7 1 6 5 4 2 3 6 7 7.5 2
15	15	2	1.5	1	0.5 1 0.5 1.5 3.5 4.5 1 4 3.5 3.5 2 3 6 7 7.5 2
16	28				0.5 1.5 0.5
17	7	3	1.5	4	0.5 1.5 0.5 0.5 3 2.5 5.5 3 0.5 1 0.5 1.5 6 7 1 6 5 4 2 3 6 7 7.5 2
18	19	3	1.5	5	0.5 1.5 0.5 2 1.5 1.5 1.5 2.5 6 7 7.5 2
19	12				0.5 1.5 1 0.5 1 0.5 1.5 4 4.5 1 4 3.5 3.5 2 3 6 7 7.5 2
20	1				0.5 1.5 1.5 0.5 2 2.5 3.5 5.5 2.5 2 5 4 8 4 1 1 0.5 1.5 6 7 1 6 5 4 2 3 6 7 8 2
21	21	2	2.5	1	0.5 1.5 1.5 1.5 1.5 2.5 6 6.5 7.5 2
22	10	2	1.5	2	0.5 1.5 1.5 3.5 2 0.5 1 0.5 1.5 5 6 1 5 4.5 4 2 3 6 7 7.5 2
23	27	4	1.5	2	0.5 1.5 2.5 0.5
24	6				0.5 1.5 2.5 0.5 0.5 4 3 5.5 3 0.5 1 0.5 1.5 6 7 1 6 5 4 2 3 6 7 7.5 2
25	18	4	1	3	0.5 1.5 2.5 0.5 3 2.5 2.5 2 2.5 6 7 7.5 2
26	26	4	0.5	1	0.5 1.5 2.5 3.5 0.5
27	17	2	2.5	3	0.5 1.5 2.5 3.5 1 3 2.5 2.5 2 2.5 6 7 7.5 2
28	5				0.5 1.5 2.5 3.5 1.5 1 4 3 5.5 3 0.5 1 0.5 1.5 6 7 1 6 5 4 2 3 6 7 7.5 2
29	25	2	2.5	4	0.5 1.5 2.5 3.5 4.5 1
30	4				0.5 1.5 2.5 3.5 4.5 2.5 2 5 4 6.5 4 1 1 0.5 1.5 6 7 1 6 5 4 2 3 6 7 7.5 2

Fig. 1. The rank-encoded suffix array $R[1..30]$ for $S[1..30] = 397235684365952201560543125671$, with $L[i]$, $B[i]$ and $D[i]$ computed for $\lg^c n = 4$. Stored values are shown in boldface.

sampld cell to obtain the contents of $R[i] = j$, extract and encode $S[j..k + m - 1]$, compare its encoding to $E(P)$ and, if they are the same, return j . This still takes $\mathcal{O}(m \log^3 n)$ time. Therefore, after we perform our binary search on R , if we find an interval of length at most $\lfloor \lg n \rfloor - 1$ which contains pointers to all the order-preserving matches for P in S (instead of an order-preserving match directly), then we can check each cell in that interval with this procedure, in a total of $\mathcal{O}(m \log^4 n)$ time.

If $R[i] = j$ is the starting position of an order-preserving match for a pattern $P[1..m]$ with $m \leq \lg^c n$ that has at most $\lfloor \lg n \rfloor$ order-preserving matches in S , then $L[i] \leq \lg^c n$. Moreover, if $R[i'] = j - 1$ then $L[i'] \leq \lg^c n + 1$ and, more generally, if $R[i''] = j - t$ then $L[i''] \leq \lg^c n + t$. Therefore, we can repeat the procedure described above and find j without ever reading a value in L larger than $\lg^c n + \lg n$ and, since each value in B is bounded in terms of the corresponding value in L , without ever reading a value in B larger than $\lg^c n + \lg n + 1/2$. It follows that we can replace any values in L and B greater than $\lg^c n + \lg n + 1/2$ by the flag -1 , indicating that we can stop the procedure when we read it. With this modification, each value in L and B takes $\mathcal{O}(\log \log n)$ bits so, since each value in D is less than $\lg^c n + \lg n$ and also takes $\mathcal{O}(\log \log n)$ bits, L , B and D take a total of $\mathcal{O}(n \log \log n)$ bits.

For example, suppose we are searching for order-preserving matches for $P = 2312$ in the string $S[1..30]$ shown in Figure 1. Binary search on R tells us that pointers to all the matches are located in R strictly between $R[16] = 28$ and $R[19] = 12$, because

$$\begin{aligned} E(S[28..30]) &= E(671) = 0.51.50.5 \\ &\prec E(P) = E(2312) = 0.51.50.52 \\ &\prec E(S[12..14]) = E(595) = 0.51.51; \end{aligned}$$

notice $R[16] = 28$ and $R[19] = 12$ are stored because 16, 28 and 12 are multiples of $\lfloor \lg n \rfloor = 4$.

We first check whether $R[17]$ points to an order-preserving match for P . That is, we assume (incorrectly) that it does; we take the first $L[17] = 3$ characters of P ; and, because $B[17] = 1.5$, we prepend a character between the lexicographically first and second, say 1.5. This gives us 1.5231, whose encoding is 0.51.52.50.5. Another binary search on R shows that $R[20] = 1$ is the last sampled element x such that $E(S[x..x+3])$, in this case 0.51.51.50.5, is lexicographically smaller than 0.51.52.50.5. Adding $D[17] = 4$ to 20, we would conclude that $R[24] = R[17] - 1$ (which happens to be true in this case) and that 0.51.52.50.5 is a prefix of $E(S[R[24]..n])$ (which also happens to be true). Since $R[24] = 6$ is sampled, however, we compute $E(S[7..10]) = 0.51.50.50.5$ and, since it is not the same as P 's encoding, we reject our initial assumption that $R[17]$ points to an order-preserving match for P .

We now check whether $R[18]$ points to an order preserving match for P . That is, we assume (correctly this time) that it does; we take the first $L[18] = 3$ characters of P ; and, because $B[18] = 1.5$, we prepend a character between the lexicographically first and second, say 1.5. This again gives us 1.5321, whose encoding is 0.51.52.50.5. As before, a binary search on R shows that $R[20] = 1$ is the last sampled element x such that $E(S[x..x+3])$ is lexicographically smaller than 0.51.52.50.5. Adding $D[18] = 5$ to 20, we conclude (correctly) that $R[25] = R[18] - 1$ and that 0.51.52.50.5 is a prefix of $E(S[R[25]..n])$.

Repeating this procedure with $L[25] = 4$, $B[25] = 1$ and $D[25] = 3$, we build a string with encoding 0.51.52.50.5, say 2341, and prepend a character equal to the lexicographically first, 1. This gives us 12341, whose encoding is 0.51.52.53.51. Another binary search shows that

$R[24] = 6$ is the last sampled element x such that $E(S[x..x + 4])$ is lexicographically smaller than 0.5 1.5 2.5 3.5 1. We conclude (again correctly) that $R[27] = R[18] - 2$ and that 0.5 1.5 2.5 3.5 1 is a prefix of $E(S[R[27]..n])$.

Finally, repeating this procedure with $L[27] = 2$, $B[27] = 2.5$ and $D[27] = 3$, we build a string with encoding 0.5 1.5, say 1 2, and prepend a character lexicographically greater than any currently in the string, say 3. This gives us 3 1 2, whose encoding is 0.5 0.5 1.5. A final binary search show that $R[8] = 14$ is the last sampled element x such that $E(S[x..x + 2])$ is lexicographically smaller than 0.5 0.5 1.5. We conclude (again correctly) that $R[11] = R[18] - 3$ and that 0.5 0.5 1.5 is a prefix of $E(S[R[11]..n])$. Since $R[11] = 16$ is sampled, we compute $E(S[19..22]) = 0.5 1.5 0.5 2$ and, since it matches P 's encoding, we indeed report $S[19..22]$ as an order-preserving match for P .

5 Conclusion and Future Work

The $\mathcal{O}(n/\log n)$ sampled values from R take $\mathcal{O}(n)$ bits and, since we replace each value in L and B greater than $\lg^c n + \lg n + 1/2$ by the flag -1 , in total L , B and D take $\mathcal{O}(n \log \log n)$ bits. Checking an unsampled position in R takes $\mathcal{O}(\log n)$ repetitions of an $\mathcal{O}(m \log^2 n)$ -time procedure (the bottleneck being the binary search, for which we extract and encode a substring of S for each step in time $\mathcal{O}(m \log n)$), and we check only $\mathcal{O}(\log n)$ unsampled positions, so we use $\mathcal{O}(m \log^4 n)$ time. Summarizing our results, we have the following theorem:

Theorem 1. *Suppose we want to support order-preserving matching in a string $S[1..n]$ for patterns of length at most $\lg^c n$ for some given constant $c \geq 1$, and that we can read any ℓ consecutive characters of S in $\mathcal{O}(\ell \log n)$ time. Then we can build an $\mathcal{O}(n \log \log n)$ -bit index such that later, given such a pattern $P[1..m]$, in $\mathcal{O}(m \log^4 n)$ time we can return the starting position of some order-preserving match for P in S , or report that none exist.*

Our approach may be applicable to a similar but older problem, parameterized pattern matching, for which we are asked to preprocess a string $S[1..n]$ over an alphabet of size σ consisting of two subalphabets, Σ_s and Σ_p , such that later, given a pattern $P[1..m]$ over the same alphabet, we can quickly find the substrings of S that can be made equal to P by changing their characters in Σ_p according to an automorphism of Σ_p . Baker [1,2] introduced this problem in 1993 and gave a kind of suffix tree to solve it, which also uses encodings of the suffixes: characters in Σ_s are left unencoded, the first occurrence of each character in Σ_p is encoded with a 0 and each other occurrence is encoded as the distance to the previous occurrence. Very recently, Ganguly, Shah and Thankachan [16] gave a succinct encoding based on the Burrows-Wheeler Transform. We believe that combining Baker's encoding with the ideas behind our index will prove the following conjecture, but we have not yet checked the details:

Conjecture 1. Suppose we want to support parameterized matching in a string $S[1..n]$ for patterns of length at most $\lg^c n$ for some given constant $c \geq 1$, and that we can read any ℓ consecutive characters of S in $\mathcal{O}(\ell \log n)$ time. Then we can build an $\mathcal{O}(n \log \log n)$ -bit index such that later, given such a pattern $P[1..m]$, in $\mathcal{O}(m \log^4 n)$ time we can return the starting position of some parameterized match for P in S , or report that none exist.

References

1. Brenda S. Baker. A theory of parameterized pattern matching: algorithms and applications. In *STOC*, pages 71–80, 1993.
2. Brenda S. Baker. Parameterized pattern matching: Algorithms and applications. *J. Comput. Syst. Sci.*, 52(1):28–42, 1996.
3. J  r  my Barbay, Meng He, J. Ian Munro, and Srinivasa Rao Satti. Succinct indexes for strings, binary relations and multilabeled trees. *ACM Trans. Algorithms*, 7(4):52, 2011.
4. Djamal Belazzougui, Adeline Pierrot, Mathieu Raffinot, and St  phane Vialette. Single and multiple consecutive permutation motif search. In *ISAAC*, volume 8283 of *Lecture Notes in Computer Science*, pages 66–77. Springer, 2013.
5. Domenico Cantone, Simone Faro, and M. Oguzhan K  lekci. An efficient skip-search approach to the order-preserving pattern matching problem. In *Stringology*, pages 22–35. Department of Theoretical Computer Science, Czech Technical University in Prague, 2015.
6. Tamanna Chhabra, Simone Faro, M. Oguzhan K  lekci, and Jorma Tarhio. Engineering order-preserving pattern matching with SIMD parallelism. *Software: Practice and Experience*, 2016.
7. Tamanna Chhabra, Emanuele Giaquinta, and Jorma Tarhio. Filtration algorithms for approximate order-preserving matching. In *SPIRE*, volume 9309 of *Lecture Notes in Computer Science*, pages 177–187. Springer, 2015.
8. Tamanna Chhabra, M. Oguzhan K  lekci, and Jorma Tarhio. Alternative algorithms for order-preserving matching. In *Stringology*, pages 36–46. Department of Theoretical Computer Science, Czech Technical University in Prague, 2015.
9. Tamanna Chhabra and Jorma Tarhio. Order-preserving matching with filtration. In *SEA*, volume 8504 of *Lecture Notes in Computer Science*, pages 307–314. Springer, 2014.
10. Tamanna Chhabra and Jorma Tarhio. A filtration method for order-preserving matching. *Inf. Process. Lett.*, 116(2):71–74, 2016.
11. Sukhyeun Cho, Joong Chae Na, Kunsoo Park, and Jeong Seop Sim. A fast algorithm for order-preserving pattern matching. *Inf. Process. Lett.*, 115(2):397–402, 2015.
12. Maxime Crochemore, Costas S. Iliopoulos, Tomasz Kociumaka, Marcin Kubica, Alessio Langiu, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Order-preserving indexing. *Theor. Comput. Sci.*, 638:122–135, 2016.
13. Gianni Decaroli and Giovanni Manzini. Compressing and indexing stock market data. *CoRR*, abs/1606.05724, 2016.
14. Simone Faro and M. Oguzhan K  lekci. Efficient algorithms for the order preserving pattern matching problem. In *Algorithmic Aspects in Information and Management*, volume 9778, 2016.
15. Paolo Ferragina and Giovanni Manzini. An experimental study of a compressed index. *Inf. Sci.*, 135(1-2):13–28, 2001.
16. Arnab Ganguly, Rahul Shah, and Sharma V. Thankachan. Parameterized pattern matching - succinctly. *CoRR*, abs/1603.07457, 2016. An updated version has been accepted to SODA 2017.
17. Pawel Gawrychowski and Przemyslaw Uznanski. Order-preserving pattern matching with k mismatches. In *CPM*, volume 8486 of *Lecture Notes in Computer Science*, pages 130–139. Springer, 2014.
18. Tommi Hirvola and Jorma Tarhio. Approximate online matching of circular strings. In *SEA*, volume 8504 of *Lecture Notes in Computer Science*, pages 315–325. Springer, 2014.
19. Jinil Kim, Peter Eades, Rudolf Fleischer, Seok-Hee Hong, Costas S. Iliopoulos, Kunsoo Park, Simon J. Puglisi, and Takeshi Tokuyama. Order-preserving matching. *Theor. Comput. Sci.*, 525:68–79, 2014.
20. Marcin Kubica, Tomasz Kulczynski, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. A linear time algorithm for consecutive permutation pattern matching. *Inf. Process. Lett.*, 113(12):430–433, 2013.
21. Rahul Shah. Personal communication, 2016.